INTRODUCING SERVICE-ORIENTATION INTO SYSTEM ANALYSIS AND DESIGN

Prima Gustiene, Remigijus Gustas
Department of Information Systems, Karlstad University, 651 88 Karlstad, Sweden
Prima.Gustiene@kau.se, Remigijus.Gustas@kau.se

Keywords: Service-oriented analysis and design, enterprise modelling, integrity of static and dynamic aspects.

Abstract: The conventional methods of information system analysis and design are not based on service-oriented paradigm that facilitates control of business process continuity and integrity. Service-oriented representations are quite comprehensible for business experts as well as system designers. It is reasonable to conceptualize a business process in terms of service-oriented events, before the supporting technical system is designed. UML design primitives abstract from the concrete implementation artefacts and therefore they are difficult to comprehend for business analysis experts. Separation of different modelling dimensions tends to draw attention away from the semantic consistency issues of various diagram types. The presented approach for service-oriented analysis is based just on three types of events: creation, reclassification and termination, which can also be used for the semantic integrity and consistency control. In this paper, the basic service-oriented constructs are defined. Semantics of these implementation neutral artefacts are analysed in terms of their associated counterparts that are used in object-oriented design.

1 INTRODUCTION

Service-oriented system analysis and design is a new emerging approach that has evolved from the object-oriented (Blaha & Rumbaugh, 2005) and component-based software engineering (Szyperski, 1998). Experience from Service-Oriented Architecture (SOA) implementation projects (Zimmerman et al., 2004) suggests that traditional information system modelling methods cover just part of required modelling notations that are currently emerging under the service-oriented analysis and design (SOAD) approaches. There are many attempts of solving this problem by defining new notations such as Archimate (Lankhorst et al., 2005), where explicit concept of service is introduced. Unfortunately, the Archimate language constructs for structural modelling of business data are underdeveloped. The lack of research on semantic integrity (Harel & Rump, 2004), (Kim et. al., 2000) among different types of diagrams is not a new fact. The consequence of analysing static and dynamic aspects in isolation is that additional quality assurance procedures are necessary for the semantic consistency and integrity control across various dimensions (Zachman, 1996).

The object-oriented methods are typically based on modelling of the use case, logical data, process, implementation and deployment views (Booch et al., 1999). Principles of integration and principles of concern separation are not clear in the conventional system analysis and design methodologies. As the concept of service is rather well understood in different domains, it could be successfully used for breaking down system functionality into coherent non-overlapping subsystems. Some information system development methodologies have argued for a single meta-model (Dori, 2002), (Gustas & Gustiene, 2004) that integrates different perspectives (Zachman, 1996). Traceability from one diagram type to another becomes a bottleneck in most system modelling approaches, if dispersed views and perspectives are defined in isolation. A fundamental problem resides in a difficulty to integrate the static and behavioural aspects of information system specifications. Service descriptions are capable to provide such integration. The problem is that comprehensive methods for service-oriented analysis are not available.

OMG (OMG Architecture Board, 2003) has extended the focus of Model Driven Architecture (MDA) to the Computation Independent Modelling (CIM), which is sometimes referred to as enterprise modelling. Most of the conventional system analysis and design methodologies, including object-oriented methods, are more relevant for logical system
design. UML modelling primitives abstract from concrete implementation artefacts and that is why such methods are more comprehensible for software designers. Computation independent descriptions are used by non-technicians, who play a key role as semantic system integrators. It is recognised that UML support for such tasks is quite vague.

SOA (Erl, 2005) represents a set of design principles (Krafling et al., 2005) that enable business processes to be analysed in terms of services. The most fascinating idea about service concept is that it can be applied equally well to organizational as well as software components. SOA is based on the assumption that business process models are composed of loosely coupled components, which are viewed as service requestors and service providers. Service propositions, requests and service provision within a value chain or within business process can be defined by using pragmatic patterns (Moor, 2005) in terms of communication actions (Dietz, 2001). To achieve ultimate flexibility, this essentially simple idea can be used at various system engineering levels (Lankhorst et al., 2005), including component-based engineering.

The objective of this study is to define the basic constructs that can be used for service-oriented analysis and semantic integration of different modelling dimensions. This paper is organised as follows. In the next section, the importance of an integrating modelling foundation for service-oriented modelling is discussed. The third section defines a set of the basic constructs for service-oriented analysis. The bridging between two ways of system analysis and design is given in the fourth section. The fifth section presents a discussion on associated problems of service-oriented events in the context of object-oriented analysis. The conclusion section outlines the perspective of service-orientation.

2 SERVICE-ORIENTED DESIGN

Services are internally characterised by state changes (Hull et al., 2003). At the same time, service semantics cannot be described independently of how these self-contained business and technical components are externally used (Moor, 2005). Integration of the internal and external behaviour (Lankhorst et al., 2005) creates big challenges for the object-oriented modelling as well as business process modelling approaches. Since the perspectives are highly intertwined, it is critical to maintain interdependency relations across multiple diagrams.

Service-orientation promotes interoperability by minimising requirements for shared understanding. Separation of concerns and integration of internal and external behaviour, which is encapsulated in a service concept, provide modelling flexibility. Business processes can be changed by replacing or recomposing services. Conceptual representations of service architectures can be used for specification of business processes in terms of organisational and technical services. Service-orientation does not exclude the object-oriented analysis and design (OOAD) point of view adopted by RUP, but rather suggests two additional layers of abstraction above it (Zimmerman et al., 2004). This idea is illustrated in figure 1.

![Figure 1: SOAD layers](image)

Figure 1: SOAD layers

Services can be understood as organizational and technical system components, which can be used by various actors to achieve their goals. A service from the information system analysis point of view is a function. It is defined by at least two flows into opposite directions between a service requester and service provider: Service Response = f (Service Request). Service providers are actors that typically receive service requests, over which they have no direct control, and transform them into responses that are sent to service requesters. This idea is illustrated in figure 2.

![Figure 2: Service as an interaction loop](image)

Figure 2: Service as an interaction loop

Services are dynamic subsystems, because outputs depend not only on inputs, but on their states (Hull et al., 2003) as well. The dynamic aspect of service can be characterized by using precondition and post-condition object classes (see next chapter). Associated classes of objects restrict service responses to the present and future inputs. Requests and responses compose together one or more service interaction loops that are crucial to understand the semantic aspects of services. A precondition object class and the input flow should
be sufficient for determining a service output flow and a post-condition object class. Enterprise system can be defined as a set of interacting loosely coupled components, which are able to perform specific services on request.

Traditional methods of information system analysis are based on the idea of dividing the technical system representations into three major parts that are known as data architecture, application architecture and technology architecture. Although there are some advantages in separation of different system specifications, there is also a fundamental problem in such way of thinking. It often results in a failure to integrate the static and dynamic aspects of different services. Since the concept of service is not used in models explicitly, such system development tradition is not able to distinguish clearly among semantic details of specifications that belong to different services. Separation of concerns about autonomic services is crucial for enterprise architecture evolution, because outdated components are often replaced.

Service-oriented analysis helps to understand why the technical system components are useful and how service descriptions are composed into the overall business process. One of the main objectives of the strategy-oriented analysis is to produce a pragmatic description. Pragmatics (Moor, 2005) provides a motivation for various configurations of service architectures and defines the "why" aspect (Zachman, 1996). It helps in building more purposeful products (Rolland, 2005). Service layer must have a capacity to integrate the static and dynamic structures of business process fragments across organisational and technical system boundaries. Most conventional methodologies are heavily centred on the class and component layers. Implementation-oriented (syntactic) layers define the technical details of a specific application. Business layer motivates and prescribes the service layer details, which are defined in terms of the basic service-oriented constructs.

3 BASIC GRAPHICAL CONSTRUCTS

Service – Oriented analysis and design is a hot research topic (Gottschalk et al., 2002). Many approaches are focusing on design of services from software components using object-oriented methods (Gustas & Jakobsson, 2004). Since object-oriented way of service modelling is computation dependent, such methods are cumbersome and just indirectly applicable for conceptual modelling of services. There are just two basic events in our service-oriented approach: creation and termination (Gustas & Gustiene, 2007). They are fundamental for the definition of reclassification event that can be understood as a communication action (Dietz, 2001). A communication action between two actors (agent and recipient) indicates that one actor depends on another actor. An instance of actor can be an individual, a group of people, an organisation, a machine, a software or hardware component, etc. The actor dependency (→) is usually viewed as a physical, information or a decision flow between two parties involved. Graphical notation of the reclassification event is presented in figure 3.

![Figure 3: Construct for representation of reclassification](image)

An action is defined as a transition (→) from the precondition object class to the postcondition object class. Fundamentally, two kinds of changes occur during any transition: removal of an object from a precondition class and creation of an object in a postcondition class. Sometimes, objects are passing several classes or states, and then are destroyed. A graphical notation of the dependency between a final state and action, that defines the termination event, can be viewed as a special case of a communication action dependency, which is not provided by a next state. The termination event is represented in figure 4.

![Figure 4: Construct for representation of termination](image)

Removal action terminates all the associations of an object. The creation action must bring all its associations of an object into existence. Graphical notation of the creation event is represented in figure 5.

![Figure 5: Construct for representation of creation](image)
A similar type of actor link that is called the strategic dependency was introduced in i* framework for early-phase requirement engineering (Yu & Mylopoulos, 1994). In our approach, the strategic dependency is considered at the same time to be an action and a communication flow. An agent initiates a flow by using an action to achieve his goal. The effect of any interaction is a reclassification, removal or creation event. Otherwise, an action is not purposeful. Composition of these three types of basic constructs is used for conceptualisation of a continuous or finite lifecycle for one or more objects in a service interaction loop.

Semantics of various kinds of static associations are defined by cardinality constraints. In the diagrams of this paper, we use a classical way of defining static associations without names of mappings in two opposite directions. We are not going any deeper into the issue of mapping names for the reason of space limitations. Graphical notation of static associations is presented in figure 6.

Many classes have common dependency links. The similarities between concepts can be shared by extracting and attaching them to a more general concept. In such a way, similar associations can be inherited by several concepts. Inheritance is often promoted as a core link to connect a specific concept to more general one. Therefore, it denotes specialisation and generalisation at the same time.

Composition is a conceptual dependency used to relate a whole to other concepts that are viewed as parts. The composition dependency is more restrictive as compared to the aggregation dependency. It is characterised by the following properties: a part cannot simultaneously belong to more than one whole of the same concept. If it does belong to more then one whole, then it must be a whole that is an instance of another concept. A part and a whole are created at the same time. Once a part is created, it can be terminated at the same time the whole is terminated. This definition is stricter as compared to a composition that is defined in the object-oriented approaches (Maciaszek, 2001). Graphical notation of the other types of the basic static dependencies is presented in figure 7.

Figure 6: Graphical notation of associations

<table>
<thead>
<tr>
<th><img src="" alt="Diagram" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1:1:1:1)</td>
</tr>
<tr>
<td>(1:0:1:*;1:1)</td>
</tr>
<tr>
<td>(1:1:*;1:1)</td>
</tr>
<tr>
<td>(1:0:1:*;1:1)</td>
</tr>
<tr>
<td>(1:1:1:1)</td>
</tr>
</tbody>
</table>

Figure 7: Graphical notation of the static dependencies

Static dependencies define complementary details for compositions of the basic event constructs, which are very important to understand semantics of service architectures. It should be noted that the presented constructs follow the basic conceptualization principle (Griethuisen, 1982) in representing only the computation independent aspects that are not influenced by possible implementation solutions.

4 FROM SERVICE LAYER TO COMPONENT LAYER

Service semantics are first conceptualised on computation independent level of abstraction. Many services can be implemented as software components. Therefore, they should be specified on a computation specific level of abstraction (component layer). Conceptual representations of service architectures are less complex as compared to computation dependent specifications. They are more comprehensible for business experts. Every communication action represents creation, termination or reclassification of one or more objects. The static dependencies predefine which object links must be created by a communication action. It should be noted that the creation and termination actions are propagated along the composition hierarchy links. Basic events of service architecture are computation neutral constructs, which helps system designers to conceptualise software components at the computation specific level of abstraction.

Conceptual representation of service architecture is defined by using one or more interaction loops. Semantics of one loop can be defined by using two constructs of basic events. Superimposition of two
interaction loops with different outcomes may result into more complex abstractions, which represent sequence, branching or synchronisation of actions. Various combinations of the static and dynamic dependencies are capable to express the main workflow control patterns (Russel et al., 2006). We are not going into any further pattern definition details for the reason of space limitations.

By matching the actor dependencies from agents to recipients, one can explore opportunities that are available to the actors. We shall illustrate interplay of three basic constructs by example of an interaction loop, which represents an exclusive choice pattern, which is often used to demonstrate semantic power of BPMN artefacts (BPMN Working group, 2004). Interaction loop between two actors (Person and CEO) is composed of the creation, termination and reclassification events, which are illustrated in figure 8.

![Figure 8: Illustration of three basic constructs](image)

This diagram illustrates how sequences of communicative actions prescribe creation, termination or reclassification operations. A person has a possibility to apply for employment by sending an application to CEO of a company. If CEO receives the application, then an object of Application and an associated object of Applicant are created (see composition link). According to the semantics of basic constructs, CEO is obliged either to employ an applicant or to reject an application. Please note that both actions predefine removal of an application object. If CEO decides to reject application, then an Applicant object is terminated. If an employee would be terminated by some action, then the association links to Position and to Employment objects must be removed.

Service architecture can be implemented as a set of loosely coupled system components. Organisational system (see figure 8) is supported by a technical system part, which can be conceptualized in terms of any number of software or hardware components. Organisational (human or business) and technical components can be represented by using an agreed set of syntactic primitives such as icons of a data file, software application, computer or human (Gustas & Gustiene, 2002). Typically, a coherent set of interactions are delegated to one independent technical component. All coherent interactions fit together for the achievement of a common goal. Interactions of one technical and two organisational components are represented in figure 9.

![Figure 9: Description of Recruitment Service](image)

The presented graphical description of the Recruitment service is consistent with the service layer specification, which is illustrated in figure 8. Coherent set of interactions are supported by one software component, which is called Recruitment Service. Please note that the interactions, which are associated with the Reject action, are not presented in figure 9 for the reason of space limitations.

The Apply action is decomposed into two operations: Send Application Data and Receive Application Data. Send Application Data is the first
operation, which is supposed to create Applicant and Application objects. The Receive Application Data operation is not just delivering Application Data flow to CEO, but also changes Application Status state from ‘Unspecified’ to ‘Received’.

5 FROM SERVICE-ORIENTED TO OBJECT-ORIENTED DIAGRAMS

Service-oriented diagram is defined in terms of creation, termination or reclassification constructs. The constructs are combined together for graphical representation of service semantics. A service-oriented diagram is computation neutral. It is more comprehensible for business experts as compared to object-oriented diagrams. Since UML has become a de facto standard for system modelling, it is a reasonable alternative for specification of software components. In this chapter, we will illustrate the bridging rules from the basic service-oriented constructs to object-oriented diagrams. For complete definition of component semantics, it is necessary to specify use case view (use case diagram), process view (activity, state and sequence diagrams) and logical data view (class diagram). In this paper, we are not dealing with the details of implementation and deployment views.

Use cases represent functionality that a software component provides by interacting with actors. Service requesters belong to the environment of a technical system part. A short guidance for specification of a use case diagram is as follows: a) Communication action is represented as a use case, b) Software component, which plays role of a service provider, defines service boundary of a technical service, c) Service requester is represented as a use case actor. Use case diagram of a Recruitment Service is illustrated in figure 10.

Service description concepts that are representing an organisational system part (see actors: Person and CEO) are not decomposed any further. Any communication action can be considered as separate function in the use case diagram. Use cases are decomposed into the component layer actions (see figure 9) by using <<include>> relationship. The <<extends>> relationship is also used between the service request and service response action, because triggering of the second is optional. According to our example, if the Apply action is triggered, then two different outcomes are possible: either Employ, or Reject. According to the service-oriented diagram, one of the successive actions must always take place. Such detail is not included into the presented use case diagram. To preserve semantic consistency, we must use expressive power of the activity diagram notation. Mutual exclusiveness of two use cases is represented by the activity diagram, which is illustrated in figure 11.

Use cases are decomposed into the component layer actions (see figure 9) by using <<include>> relationship. The <<extends>> relationship is also used between the service request and service response action, because triggering of the second is optional. According to our example, if the Apply action is triggered, then two different outcomes are possible: either Employ, or Reject. According to the service-oriented diagram, one of the successive actions must always take place. Such detail is not included into the presented use case diagram. To preserve semantic consistency, we must use expressive power of the activity diagram notation. Mutual exclusiveness of two use cases is represented by the activity diagram, which is illustrated in figure 11.

Semantics of a use case can be represented by using sequence and activity diagrams. We will limit the process view examples just to activity diagrams. The object-oriented operations, which define a use case, can be elicited from the service-oriented diagrams. A method for implementation of the Apply action is defined by using UML activity diagram, which is presented in figure 12.

According to our example, the method of the Apply use case must include two interface operations: Send Application and Receive Application. Send Application operation should
trigger Create Applicant and Create Application operations. According to semantics of service-oriented events, Receive Application operation is executed together with a Change Application Status operation that is initialising state of an Application object with the status ‘Received’.

Use case Employ consists of two interface operations: Employ Applicant and Receive Employment. The remaining domain operations are predefined by the service description as well. The corresponding UML activity diagram is represented in figure 13.

The precondition and postcondition object classes that are defined by the service description can be implemented in a number of ways. In the presented example, all service description classes are viewed as independent UML domain classes. Corresponding domain class operations are prescribed by the reclassification, creation and removal events. Employ Applicant is a reclassification event that creates Employee object and removes Applicant object. Since object-oriented approach does not support the dynamic reclassification, it can be implemented by firstly creating a new object in the postcondition class and secondly - terminating an object in the precondition class.

Conceptual representation of Recruitment Service prescribes two types of interface classes – one for a Person and one for CEO. For instance, Send Application and Receive Employment operations must be included into Interface Person class. Receive Application and Employ Applicant operations are defined in the Interface CEO class. Class diagram is illustrated in figure 14.

![Figure 13: Method of the Employ use case](image)

![Figure 14: Class diagram](image)

If CEO decides to employ an applicant, then Employ Applicant operation is triggered in the Interface CEO class. According to the presented service description, Employ Applicant action requires both to Create Employee and to Delete Applicant. Creation of a new Employee object in turn requires creation of an Employment class object as well. That is why Create Employee operation is defined in a sequence with the Create Employment operation. Since an Applicant is composed of an Application, the creation of an Applicant object is synchronised with creation of an Application object (see Delete Application, Delete Applicant and Create Applicant and Create Application operations in both activity diagrams) as well. The communication loop is completed, when a person receives Employment Data. This information flow is provided by the Receive Employment operation, which is placed in the Interface Person class. As it is prescribed by service-oriented diagram, Receive Employment is executed in sequence with Change Employment Status operation.

A method for implementation of the Reject action (see figure 11) can be designed by using the same way of thinking. If CEO rejects application, then the Reject Application operation (InterfaceCEO class) should be triggered. Receive Rejection operation in the Interface Person class must be synchronised with the Delete Applicant operation, which terminates an Applicant. Since an Applicant is composed of an Application, these two objects are supposed to be terminated simultaneously. In general, if the termination event takes place, then all objects in
more specific classes are terminated as well (see inheritance links). This rule is not relevant for the objects of more generic classes. If an object is terminated in a more specific class, then objects of the more generic classes are still preserved. For instance, even if an Applicant (and Application) object is terminated, a Person object is not lost and therefore the attribute values of SS Number and Name are preserved.

6 INTERNAL STATE CHANGES

Use case, sequence and activity diagrams are used to specify interaction model (Booch et al., 1999). Both interaction and state diagrams are necessary to fully define a process view. State changes are typically specified by using a finite state machine. It describes the sequences of operations that occur in response to various events. Therefore, state changes are complementary to interaction diagrams. Since service-oriented diagrams are combining both aspects together, they can be used for semantic integrity control of object-oriented diagrams. UML diagrams are projecting interaction and state-transition aspects into different diagram types. Coherence of both aspects is crucial in order to facilitate reasoning about business process composition from services. State changes represent structural changes of objects, which can not be analysed in isolation from the interactions. UML individual diagram types are clear enough, but integrated semantics among models is missing. UML diagrams alone are difficult to apply for semantic integrity control and for consistent business logic alignment with the system design for making both organisational and technical system parts more effective.

The basic service-oriented constructs are capable to express component interactions together with the following six types (Martin and Odell, 1998) of object-oriented events:
- creation a new object,
- termination of an object,
- classification of object into a subclass,
- declassification of an object from subclass into a superclass,
- connection of objects,
- disconnection of an existing association between two objects.

Object-oriented diagrams represent such events in a variety of ways. For instance, creation and termination operations can be represented by transitions from initial state and to final state in a finite state machine. Connection and disconnection events correspond to operations that are associated with ordinary state transitions. Classification and declassification events can be implemented by using a sequence of creation and termination operations. For the sake of simplicity, we will use object flows to represent object manipulation events. Object flow diagrams are useful to define dynamic relationships between an operation and its input/output objects represented as arguments. The object flows are used together with the UML activity diagrams. A diagram showing operations and object flows with states has most of the advantages of an activity diagram without most of their disadvantages (Blaha & Rumbaugh, 2005). It integrates moving data and control flow in the same diagram type. An input arrow to an operation or output arrow from an operation represents a control flow.

Termination event can be expressed by using the Access and Removal operation, which consumes an input object in the final state. It is represented in figure 15.

![Figure 15: Graphical representation of termination](image)

The object flow is created by an access operation, since it returns the object, which is removed by a next operation. The creation event can be defined by using a creation operation, which produces an output object in an initial state. A state can be defined as a collection of associations an object has with other types of objects and its attribute values. To perform a state change, an object needs to be accessed. Semantics of a state change can be defined as a reconnection, which is understood as an update of one or more attribute values. A state-transition diagram or an object flow diagram is able to define an object progression from one state to another. A reconnection event pattern is represented in figure 16.

![Figure 16: Graphical representation of reconnection](image)

Service-oriented events are able to express semantics of disconnecting and connecting operations by using the reclassification. Reclassification action is unique. It specifies precondition and postcondition object class associations, which are sufficient to visually recognise and understand the details of disconnections and connections. The reconnection event pattern, which is defined by object flow diagram, is not unique. It cannot be visually recognized by a system designer. The same pattern can be used for representation of the other object manipulations such as connection and disconnection. For instance, the graphical representation...
of a disconnection event will look like the reconnection event. The connection event pattern has the same structure, except that a different name for the connection operation will be used. The disconnection event pattern is represented in figure 17.

![Figure 17: Graphical representation of disconnection](image)

Any state change in a state-transition diagram can be interpreted as a reconnection event, which is represented by a sequence of one disconnection and one connection event. A sequence of two events requires an intermediate state, which makes no sense for a person, who has no or even little expertise in the area of object-oriented design. Such artificial states are implementation oriented details, which add additional complexity. An intermediate state is redundant and it cannot be validated by a business expert.

The static aspects are crucial for complete understanding of the service-oriented event semantics. On the other hand, the consequences of events that cause changes in object structure are difficult to trace by using different types of object-oriented diagrams. The mechanism of semantic integrity control is not clear. Exactly the same object-oriented pattern can be used for the semantic representation of different types of service-oriented events. That is why using object-oriented diagrams for conceptual modelling of service architectures is very difficult.

Object-oriented approaches are not supporting the dynamic classification. The termination and creation operation can be used for implementation of a declassification and classification. Reclassification is a compound event that is viewed as a simultaneous declassification and classification of an object to another class. Semantics of reclassification is quite comprehensible for business experts. Nevertheless, reclassification event has no easily recognisable counterpart in object-oriented models. The method of reclassification, declassification and classification from Class1 to Class2 can be defined in the same way. It is shortly characterised as follows:
- Creation of an object in Class2 by copying all attribute values from an object in Class1 to the newly created object in Class2,
- Disconnection of all associations pointing to the old object and connection of them to the new one in Class2,
- Removal of the old object in Class1.

The described sequence of operations artificially adds complexity by multiplying a number of states, which cannot be justified from the service-oriented analysis point of view. Higher complexity of state diagrams increases the complexity of system analysis phase in general.

The conclusion is quite obvious: UML notation is inconvenient for the implementation neutral service-oriented analysis. The problem is that the same object-oriented constructs cannot be efficiently used in the analysis phase. Usage of implementation dependent constructs increases system specification complexity. It creates difficulties in the process of validation of service architectures by business analysis experts and verification of object-oriented diagrams by design experts. The difficulty resides in failures to recognise the noteworthy changes that are caused by service events. It makes overall enterprise architecture prone to mistakes, which are introduced by inconsistencies, discontinuities and ambiguities.

7 CONCLUDING REMARKS

Service-oriented analysis is based on the assumption that business process models are composed of loosely coupled components, which are viewed as service requesters and service providers. Service propositions, requests and service provision within a value chain are defined by using a set of basic events. Semantics of service-oriented events were explained in object-oriented design terms. We concluded that UML notation is inconvenient for systematic analysis of the service-oriented events. It creates difficulties in validation of the diagrammatic solutions by business process analysis experts. Disparate diagrams are prone to inconsistencies, discontinuities and ambiguities. The presented way of analysis and design is based on one semantic model. Service-oriented constructs are quite comprehensible. They can be communicated among business experts and designers more effectively than a set of various types of implementation dependent object-oriented diagrams.

Implementation bias of many information system modelling methods is a big problem, since the same implementation oriented foundations are applied in system analysis phase, without rethinking these concepts fundamentally. Conceptual representations of service architectures define computation independent aspects of business processes, which are not influenced by the implementation dependent solutions. Our modelling approach is aiming at an engineering process that is based on one model, which is used to conceptualise service oriented architecture before the supporting technical system (component and class layers) is defined.

We have demonstrated a way of bridging from the conceptual representations of service
architectures to object-oriented diagrams. The basic service-oriented constructs redefine equivalent semantic details that were used for elicitation of object-oriented operations. One obvious advantage of service-oriented modelling is that it provides an integrated view to various types of UML diagrams. Computation independent representations are easier to comprehend for business process analysis experts as well as for object-oriented system designers. Moreover, conceptual representations of service architectures provide principles for separation of concerns and organising system functionality into coherent non overlapping subsystems, which are specified by using object-oriented diagrams. Traditionally, object-oriented diagrams are used in system analysis as well as in system design phases. It makes difficult for developers to identify noteworthy semantic difference among various states of objects. Our experience in analysing system specifications by using computation independent notation demonstrates that service-oriented events are more comprehensible. Service-oriented diagrams have no implementation bias and therefore they bridge a communication gap among system designers and business analysis experts more effectively.

REFERENCES